Classification and Regression: from linear and logistic regression to neural networks
FYS-STK3155 - Project 2

Gulla Serville Torvund and Karl Henrik Fredly
(Dated: November 14, 2020)

In this project we will look at the performance of traditional statistical methods versus the performance of a feed forward neural network on both classification and regression problems.

We will look at the regression problem of fitting the Franke function with a small number of data points. We will find that the best performing model for this task is the analytical Ridge regression model, with a fourth order polynomial that is a good guess for the shape of the Franke function, only getting a mean squared error (MSE) of 0.014. Ridge regession using stochastic gradient descent underperforms with a MSE of 0.02, as it only approaches the minimum the analytical ridge finds directly. The neural network slightly underperformed with a MSE of 0.15, as our limited search was unable to find a model that did not suffer from overfitting on such a small dataset. The neural network from tensorflow saw similar results.

The classification problem we will look at is classifying 8x8 pixel grayscale images of handwritten digits from the MNIST dataset. Here, the generality of neural networks led to a test accuracy of 0.97, while our own logistic regression model, and the logistic regression model from sklearn both only achieved a test accuracy of 0.96. Here, the simpler models had no advantage from their "good guesses", as the neural network only added flexibility to the model.

I. INTRODUCTION

During the last decade, we have year by year experienced a great improvement and an increased usage of neural networks. The 2012 revolution, sparked by the ImageNet challenge, shed light on the importance of data and the power of deep learning. We intend in this project to dip our toes into the deep waters of neural networks and supervised learning, diving into the functionality of feed forward neural networks.

Naturally, we need something to compare the performance of our neural network to, which will be linear and multinomial logistic regression. This brings up another important aspect of this project, namely being aware of the robustness of traditional statistical methods. It can be tempting to join the AI hype, applying neural networks wherever. Still, we observe that statistical methods outperform neural networks in certain areas, and, as we do in this project, these methods must be addressed.

In the methods section we will present the different regression methods and the theory behind them, as well as our way of optimizing and evaluating the models. In particular, we will present the equations underlying feed forward neural networks. The regression and classification problem will also be introduced here.

In the results section we will present the performance of our different methods with different learning rates and regularization parameters. The models shown will be the best ones we found in our limited search.

Finally, we will analyze our results and discuss what they mean for the different methods in the discussion and conclusion sections.

All code used to calculate the results in this report can be found here: https://github.com/KarlHenrik/FYS-STK-Gruppe/tree/master/Project2. For this project we implemented stochastic gradient descent (SGD) with and without momentum, ridge regression with and without SGD, a feed forward neural network for regression and classification with a flexible number of hidden layers, and a logistic regression model.

II. METHODS

We want to compare the performance of ridge and logistic regression with the performance of a feed forward neural network. Before we do that, however, we will discuss how these methods function, and how we evaluate them.

A. Linear Regression

We will in this project, as in project 1, use the linear regression method Ridge regression. This method will be used to fit polynomials to the Franke function, which will be described later.

Linear regression tries to model the linear relationship

$$\mathbf{y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

between the dependent variable $\mathbf{y}$ and the independent variable X by finding the parameter $\boldsymbol{\beta}$ which "best" satisfies the equation.

Normally one would use the regression methods' cost function and from it find an analytical expression for the parameters. In our case, however, we will use the gradient descent method to optimize the parameters. With this method the parameters are updated iteratively by minimizing the cost function using its gradient. We will

explain the principles of gradient descent more closely in the following section.

To measure the performance of the linear model, we will use the mean squared error (MSE)

$$MSE(\mathbf{y}, \tilde{\mathbf{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2,$$

where y_i is the i-th element of $\mathbf{y}$, which we want to approximate with the prediction given by our model $\tilde{\mathbf{y}} = X\boldsymbol{\beta}$, whose i-th element is $\tilde{y}_i$. The MSE measures the mean of the squares of the differences between the values in the given dataset and those predicted by our model. A low MSE indicates a good fit, as our model closely reproduces the given data.

The MSE will be our cost function. Ridge regression has a regularization parameter λ that prevent overfitting by "punishing" large regression coefficients. Its cost function therefore has an extra term:

$$\begin{aligned} C_{Ridge}(\hat{\boldsymbol{\beta}}) &= \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 + \lambda \sum_{i=0}^{n-1} (\hat{\beta}_i)^2 \\ &= ||\boldsymbol{y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}||_2^2 + \lambda ||\hat{\boldsymbol{\beta}}||_2^2. \end{aligned}$$

This cost function is the same as the MSE when $\lambda = 0$ and can therefore also be applied to OLS.

The gradient of the cost function is the one we will use for gradient descent. This is given by

$$\frac{\partial C}{\partial \hat{\boldsymbol{\beta}}} = X^T(\mathbf{y} - \tilde{\mathbf{y}}) + 2\lambda\hat{\boldsymbol{\beta}}$$

Together with this way of evaluating the model, we use the resampling technique cross-validation (CV). This gives us a more robust measure of the average performance of our model and ensures us that the result is not victim of especially good or bad initial values of the parameters. More precisely we use the k-fold CV. We will resample the data by splitting it up in k equally sized subsets of data, and use k-1 subsets as training data and the remaining subset as test data. This is repeated k times so that each subset gets to be the test data one time.

B. Stochastic Gradient Descent (SGD)

We want our models to have the best possible fit. That is, we want their cost functions to be as close to zero as possible. A way to approach that is through gradient descent, which iteratively finds the minimum of our models' cost functions. This is done by taking steps proportional to the negative gradient of the cost function in a current position. Figure 1 shows the most basic intuition behind the method.

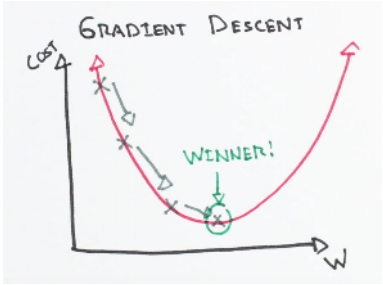

Figure 1. Illustrating the basic concept of gradient descent.

A problem with this method is that we run the risk of ending up in a local minimum instead of a global one. Actually, we won't know if we end up in a global or local minimum, but either way we wish it to be a minimum that makes our model "good enough". We can to some degree handle this problem through the *learning rate*, often denoted as η. Choosing η, we need to find the golden mean between a small enough η so that we don't skip a good minimum and a large enough η so that we at all reach the minimum.

The basic formula for an updated parameter $\boldsymbol{\beta}$ using gradient descent thus reads as follows:

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} - \eta^{(n)} g^{(n)}$$

with $g^{(n)}$ being the gradient of the cost function at the current point.

As this section's title suggests, we will in this project more specifically use *stochastic* gradient descent, commonly referred to simply as SGD. Instead of calculating the actual gradient from the entire data set, SGD uses an estimated gradient calculated from a random subset of the data. This way, we reduce the computational burden high-dimentional optimization problems entails, and make our algorithm faster. Of course this is traded for accuracy considering our gradient being a stochastic approximation.

A further extension of SGD is *SGD with momentum*. The momentum method remembers and uses the previous update of the parameters in addition to the current gradient to update the parameters:

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} - \left(\eta^{(n)} g^{(n)} + \alpha \eta^{(n-1)} g^{(n-1)}\right)$$

with α regulating how much the previous update should matter considering the next update.

Empirically, we know that most things have a tendency to keep moving in the same direction that they did a moment earlier. Momentum SGD can therefore help to accelerate SGD in the relevant direction, preventing the unnecessary oscillation ordinary SGD can lead to in curvy areas.

C. Multinomial logistic regression

Logistic regression is used to model the probability for an event or category to be true. That is, this method is used when the target (the dependent variable) is categorical - unlike cases for linear regression which are numerical.

The simplest logistic regression is the binomial logistic regression, which deals with binomial problems such as win/lose and benign/malignant. This can be extended to multinomial logistic regression, also called softmax regression, which models several classes. We will in this project be using the latter to determine which handwritten digit an image contains.

Our goal is to predict the target t from an input $\mathbf{x}$. Using softmax regression, we will then transform the input of dimensionality p, determined by number of features, for example number of pixels, into an output vector $\mathbf{y}$ of dimensionality c where c is the number of categories. The ultimate goal is that $\mathbf{y}$ will consist of all zeroes but one "1" somewhere (a so called one-hot), indicating what class $\mathbf{x}$ belongs to. So we wish to predict the probability $P(\mathbf{y} \in c_k|\mathbf{x})$ that the output $\mathbf{y}$ is in a certain class c_k, given the input provided. This is done through a number of steps.

First, we run a linear transformation on the data similar to linear regression. This is done to hopefully combine relevant features of the data in a way that categorizes the data somewhat. The raw data $\mathbf{x}$ is therefore linearly transformed into $\mathbf{z}$ by multiplying it with weights $\mathbf{w}$ and adding biases $\boldsymbol{\beta}$ so that $z = wx + \boldsymbol{\beta}$. Since we want our output vector to be c-dimensional, $\mathbf{z}$ must also be. Consequently, $W \in \mathbb{R}^{p\times c}$ and $B \in \mathbb{R}^{c\times 1}$.

We then send the vector $\mathbf{z} \in \mathbb{R}^{c\times 1}$ through the softmax function

$$Softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

which processes the coefficients in $\mathbf{z}$ and returns a probability vector $\mathbf{y}$. The placement of the vector coefficient with the largest magnitude in $\mathbf{y}$ determines which class the input data belongs to.

An optimal $\mathbf{y}$ would contain only one "1" and otherwise zeroes, signaling one hundred percent certainty of what class $\mathbf{x}$ belongs to. This is however rarely the case, and certainly not for a random set of weights and biases, which we begin with. We therefore need a cost-function to update the parameters and optimize our model. The softmax cross entropy, also called the negative log likelihood loss

$$C = CE(t, y) = -\sum_i t_i log(y_i)$$

is a standard choice as cost-function in softmax regression. Since the t, the label, is a one-hot, we can discard the elements of the summation which are zero due to target labels. Cross entropy will therefore simply return the negative logarithm of y_c where c is the positive class. The larger the certainty is for the correct class, the smaller the loss ($log 1 = 0$).

In our project we use gradient descent as means for updating the parameters. We will then need the gradient of the cost function with respect to $\mathbf{z}$. Using the chain rule with the derivative of the softmax function, this expression turns out to be very simple and elegant:

$$\begin{aligned}\nabla_{zi} CE(t, y) &= \nabla_{zi}\left(-\sum_i t_i log(y_i)\right) \\ &= t_i - y_i\end{aligned}$$

For evaluation we have chosen a way of measuring the accuracy score for our model that both ensures that the model won't have any "lucky guesses" when it has no real preferred prediction and doesn't require absolute certainty in its prediction: We label a prediction as correct if the model has at least 50 % certainty of the input belonging to the correct class. Basically, the "correct" element in the probability vector $\mathbf{y}$ has to be of magnitude 0.5 or larger.

The accuracy score is then the fraction of the test data the model classifies correctly.

We write our own code for the softmax regression, as well as using Scikit-Learn's logistic regression functionality for comparison.

D. Feed Forward Neural Network

A feed forward neural network is a simple type of neural network used for predictive modeling, among other things. The network consists of neurons, weights, biases and activation functions, which hopefully are able to turn an input into a useful output. We will here give a very brief overview of how a feed forward neural network is structured, and how backpropogation gives us the derivatives we need to improve our network with gradient descent. We choose to use matrix notation to make the equations more readable, and since it is closer to how we implement the equations in our code [1]. Note that this can make the intricacies of the network structure harder to grasp, and we therefore recommend the book by Hastie et al.[2] for a more thorough introduction to neural networks.

1. *Network Structure*

The network has an input layer, as shown in figure 2, where some input vector of data a^0 is fed. Each "input node" will hold a single number from this vector. These numbers are then fed to the next (hidden) layer of nodes, as shown by the arrows, with each number being multiplied by a weight, also represented by the arrows. When these numbers reach the next layer they collectively form

a new vector $z^1 = a^0 w^1 + b^1$, where w^1 is the matrix of weights corresponding to the first set of arrows in figure 2, and b^1 is a vector of the biases belonging to each neuron in the hidden layer.

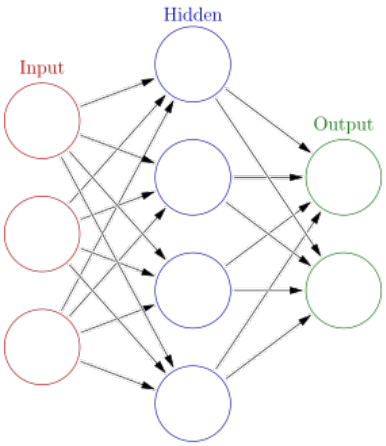

Figure 2. Example feed forward neural network with one hidden layer.

This z^1 is then fed through an activation function f belonging to the hidden layer so that the final values of the nodes in the hidden layer become $a^1 = f(z^1)$.

These values are again fed forward to the next hidden layer, or output layer (depending on the number of layers in the network). They are again multiplied by the weights, added to the biases, and fed through the activation function of the next layer. This is repeated until the final layer of the network, the output layer, is reached. Then, the values a^L are the very outputs of the network.

When training and evaluating our network we use a cost function to compare the output of the network with the real output from our data. This could be the mean squared error in the case of a regression model, or the cross entropy in the case of classification.

2. *Backpropogation*

The backpropogation algorithm uses repeated applications of the chain rule to calculate the derivatives of the cost function of our network with respect to the different weights and biases. Using these derivatives we find how we need to change these weights and biases to minimize the cost function.

For the output layer we define a δ-term.

$$\delta^L = \frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L} = \frac{\partial C}{\partial a^L}\frac{\partial f(z^L)}{\partial z^L}$$

Where we use that $a^L = f(z^L)$.

For every other layer l we get an inductive definition for similar δ-terms using every layer later in the network than l. To calculate these we need every previous δ-term, and we "propagate backwards" through the network to calculate them.

$$\delta^l = \frac{\partial C}{\partial z^l} = \frac{\partial C}{\partial z^{l+1}}\frac{\partial z^{l+1}}{\partial a^l}\frac{\partial a^l}{\partial z^l} = \delta^{l+1}(w^{l+1})^T\frac{\partial f(z^l)}{\partial z^l}$$

Where we used the first of these three results

$$\begin{aligned} z^{l+1} = a^l w^{l+1} + b^{l+1} &\Rightarrow \frac{\partial z^{l+1}}{\partial a^l} = w^{l+1} \\ &\Rightarrow \frac{\partial z^{l+1}}{\partial w^{l+1}} = a^l \\ &\Rightarrow \frac{\partial z^{l+1}}{\partial b^{l+1}} = 1 \end{aligned}$$

For all layers l we can then calculate the derivative of the cost function with respect to the weights and biases of that layer. We use the results we found above to find that

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial w^l} = (a^{l-1})^T\delta^l$$

and

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial z^l}\frac{\partial z^l}{\partial b^l} = \delta^l$$

These equations let us find the gradients of the cost function with respect to all the weights and biases of the network. In our implementation we calculate the gradient for multiple input data at the same time by letting the input a^0 be a matrix with each row being a separate input. We then take the mean of all gradients before returning the final gradient.

3. *Activation and cost functions*

A neural network consists of many parts which must be selected depending on the task at hand. In the case of the regression problem, we found the mean squared error to be a good choice as our cost function.

$$C(t, a^L) = MSE(t, a^L) = \frac{1}{2}\frac{1}{n}\sum_{i=0}^{n-1}(t_i - a_i^L)^2$$

where t is the target, and a^L is the output of the final layer in our network, our prediction. For the backpropogation we need its derivative, which is given by

$$\frac{\partial C}{\partial a^L} = (t - a^L)$$

Some popular activation functions and their derivatives for the layers in our network are

$$Sigmoid(z) = \sigma(z) = \frac{e^z}{1+e^z}, \sigma'(z) = \sigma(z)(1-\sigma(z))$$
$$Linear(z) = z, Linear'(z) = 1$$
$$ReLU(z) = \begin{cases} z & \text{for z} > 0 \\ 0 & \text{else} \end{cases}$$
$$ReLU'(z) = \begin{cases} 1 & \text{for z} > 0 \\ 0 & \text{else} \end{cases}$$
$$LeakyReLU(z) = \begin{cases} z & \text{for z} > 0 \\ 0.01z & \text{else} \end{cases}$$
$$LeakyReLU'(z) = \begin{cases} 1 & \text{for z} > 0 \\ 0.01 & \text{else} \end{cases}$$
$$Softmax(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

For the classification problem we found the cross entropy to be a good choice of cost function

$$C(t, a^L) = CE(t, a^L) = -\sum_i t_i log(a_i^L)$$

The cross entropy function works well in tandem with the softmax or sigmoid activation functions in the output layer as in both cases the derivatives come together to simplify to

$$\delta^L = \frac{\partial C}{\partial a^L}\frac{\partial f(z^L)}{\partial z^L} = (t - a^L)$$

For classification we will use the accuracy score as defined in the previous section to evaluate the performance of our model.

E. Our regression problem

Our regression problem will be to approximate the two-dimensional function called the Franke function. It is a typical choice for testing linear regression methods. The Franke function is given by

$$\begin{aligned} f(x,y) = &\frac{3}{4}\exp\left\{\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right)\right\} \\ &+ \frac{3}{4}\exp\left\{\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right)\right\} \\ &+ \frac{1}{2}\exp\left\{\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right)\right\} \\ &- \frac{1}{5}\exp\{(-(9x-4)^2 - (9y-7)^2)\}, \end{aligned}$$

and is in our case defined for $x, y \in [0, 1]$. Figure 3 shows the Franke function.

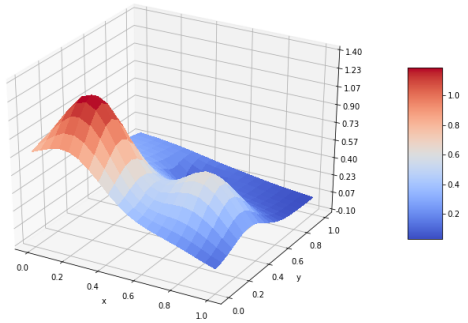

Figure 3. A 3D plot of the Franke function, which we will use to test our regression methods.

F. Our classification problem

To test how our logistic regression and neural network performs on classification, we will be using the MNIST database (Modified National Institute of Standards and Technology database). This is a large database of handwritten digits and contains 60,000 training images and 10,000 testing images. The images consists of 28x28 greyscale pixels. We will in this project only use the 1797 8x8 reshaped images from this dataset available from the sklearn python library. One of the reshaped images is shown in figure 5.

Figure 4. Sample images from MNIST test dataset.

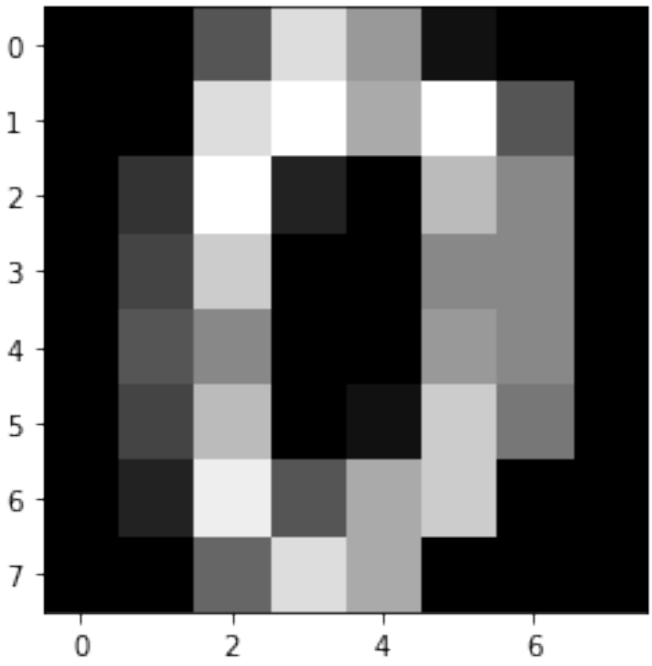

Figure 5. The handwritten digit "0" in 8 x 8 pixels.

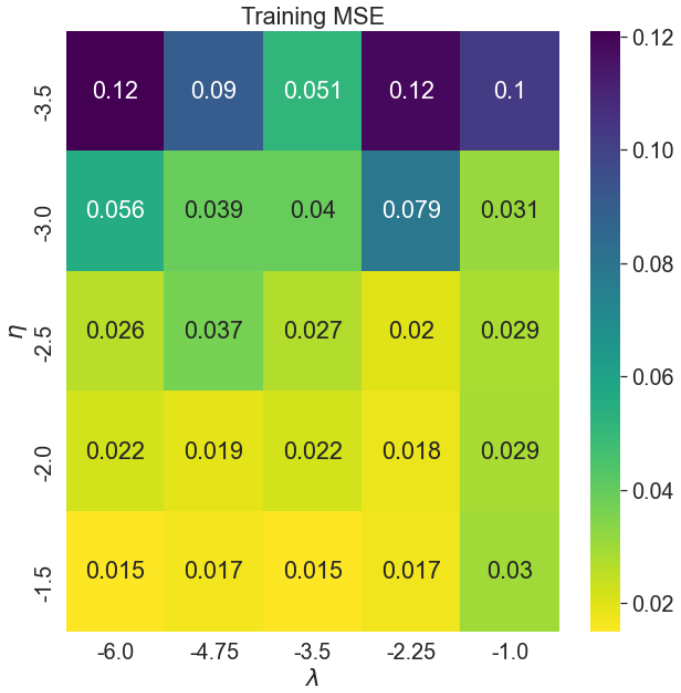

Figure 6. The training error for ridge regression with stochastic gradient descent. A polynomial or order 4 was used, a batch size of 10 and 1000 epochs. 5-fold cross validation was used on 100 datapoints of the Franke function to find the mean squared error of the fit.

III. RESULTS

A. Regression

We start by looking at the results of ridge regression. A grid search of λ values from 10^{-10} to 1 and polynomial orders from 3 to 18 found that an order of 4 and λ of 10^{-4} gave the best test error for a ridge model using the analytical formula to minimize the cost function. This model was found to have a mean squared error of 0.014 using cross validation.

We now use ridge regression with stochastic gradient descent to see how it compares to the analytical solution. We use a polynomial of order 4, and find the training(fig. 6) and test(fig. 7) error for different values of λ and for different learning rates.

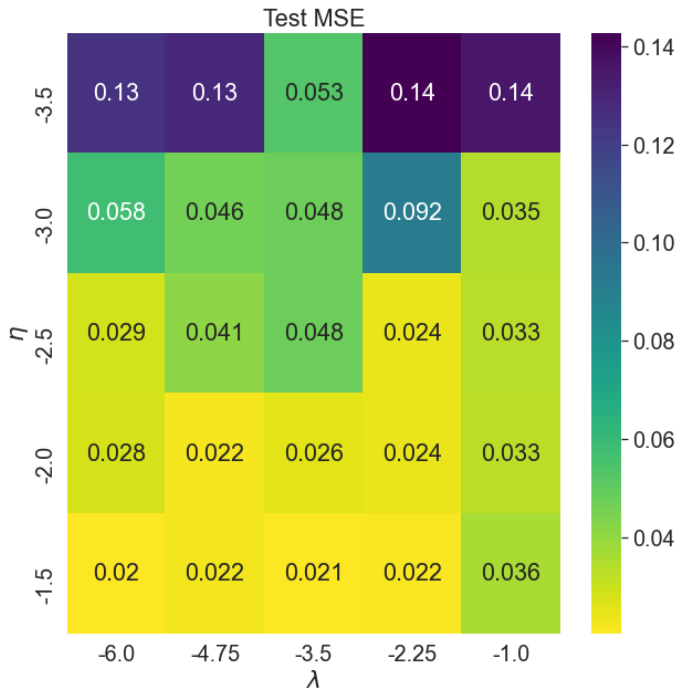

Figure 7. The test error for ridge regression with stochastic gradient descent. A polynomial or order 4 was used, a batch size of 10 and 1000 epochs. 5-fold cross validation was used on 100 datapoints of the Franke function to find the mean squared error of the fit.

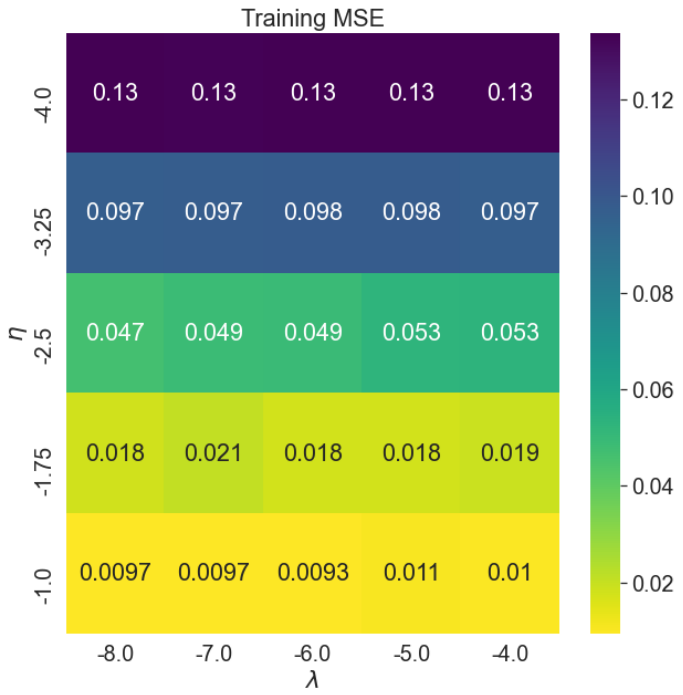

Figure 8. The training error for regression with our feed forward neural network. The model here used two hidden layers with 20 nodes and LeakyReLU activation. The linear activation function in the output layer. Stochastic gradient descent with momentum was used. 5-fold cross validation was used on 100 datapoints of the Franke function to find the mean squared error of the fit.

We tested the performance of our neural network with different numbers of hidden layers, nodes per layer, activation functions, regularization parameters, learning rates, and with both momentum and normal stochastic gradient descent, with different batch sizes and number of epochs. There are no general best choices for these values, as it depends on the problem at hand.

The best performing parameters we found in our limited search of the infinite hyperparameter space were two hidden layers with 20 nodes and LeakyReLU activation. The linear activation function in the output layer. Momentum gradient descent with a learning rate of 0.1, and a regularization parameter of 10^{-5} as shown in figure 9. The model trained with these parameters achieved a test accuracy of 0.015, using 5-fold cross validation on 100 data points.

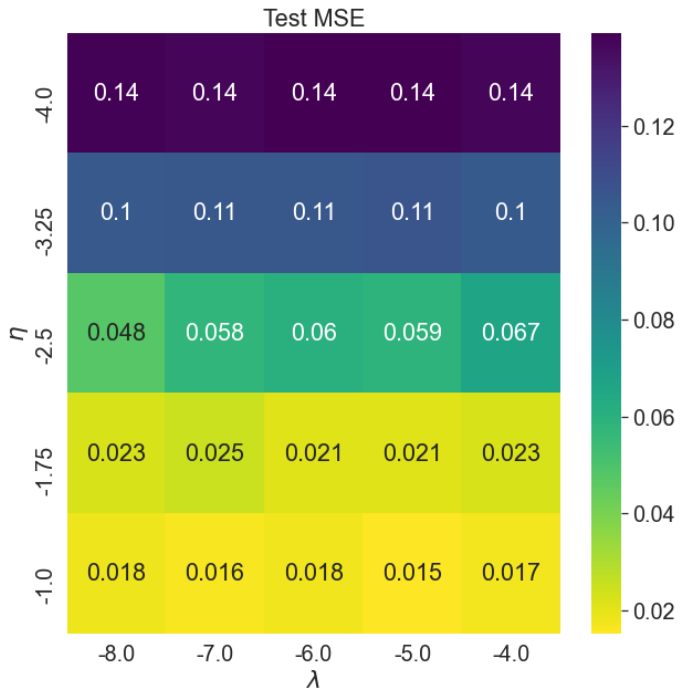

Figure 9. The test error for regression with our feed forward neural network. The model here used two hidden layers with 20 nodes and LeakyReLU activation. The linear activation function in the output layer. Stochastic gradient descent with momentum was used. 5-fold cross validation was used on 100 datapoints of the Franke function to find the mean squared error of the fit.

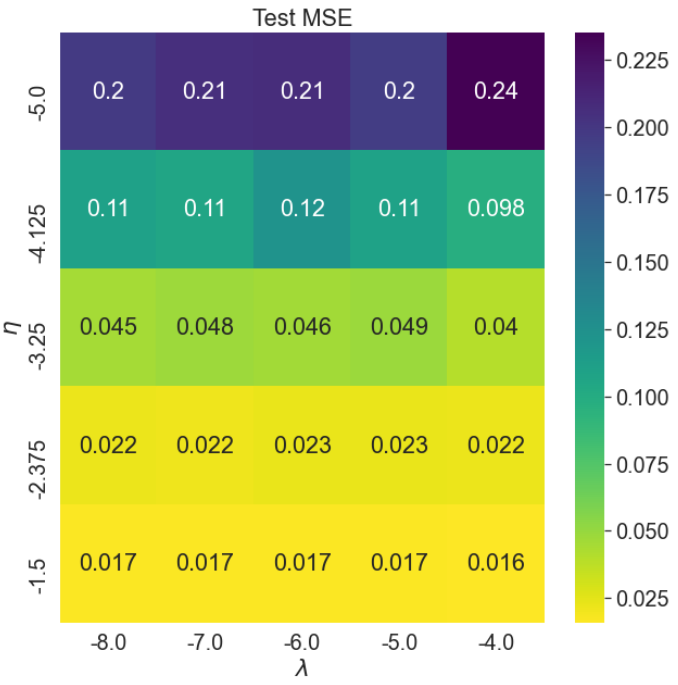

Figure 10. The test error for regression with our the tensorflow/keras Sequential model. The model here used two hidden layers with 30 and 20 nodes, both with ReLU activation. The linear activation function in the output layer. Normal stochastic gradient descent was used. 5-fold cross validation was used on 100 datapoints of the Franke function to find the mean squared error of the fit.

We used the Sequential model from tensorflow/keras to compare with our own implementation of a feed forward neural network. For this implementation, the best performing parameters we found were two hidden layers, the first with 30 nodes and the second with 20. Both with ReLU activation. The linear activation function in the output layer. Stochastic gradient descent with a learning rate of $10^{-1.5} \approx 0.3$. A regularization parameter of 10^{-4}. The model trained with these parameters achieved a test accuracy of 0.016, using 5-fold cross validation on 100 datapoints.

Finally, we also ran some of our tests with 1000 data points. The analytical ridge got a MSE of 0.012, and our neural network with the same structure was able to achieve the same MSE with the same optimal parameters.

B. Classification

The training and test accuracy of our multinomial logistic regression model is shown in figure 11 and 12. The model was trained and tested with 5-fold cross validation. We used stochastic gradient descent with 100 epochs and a batch size of 10.

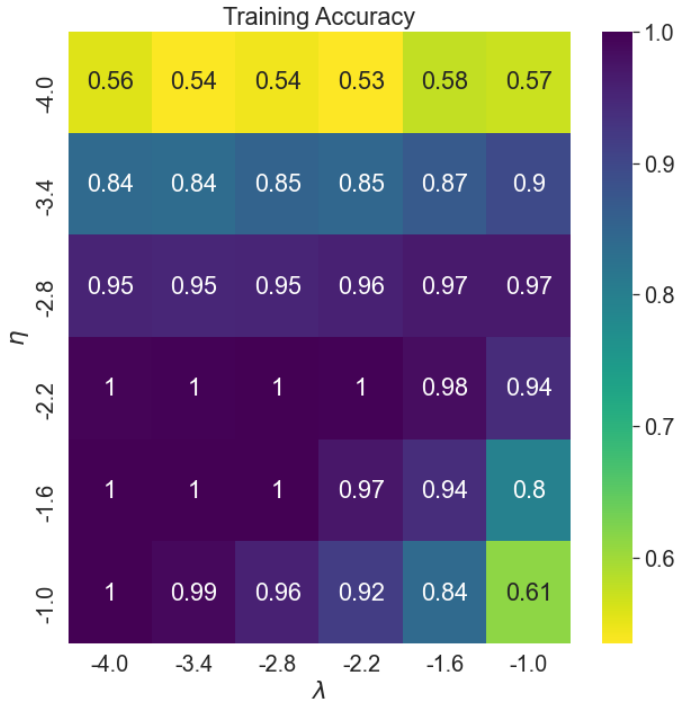

Figure 11. The training accuracy for multinomial logistic regression on the 8x8 MNIST digits. The model was trained and tested with 5-fold cross validation. We used stochastic gradient descent with 100 epochs and a batch size of 10.

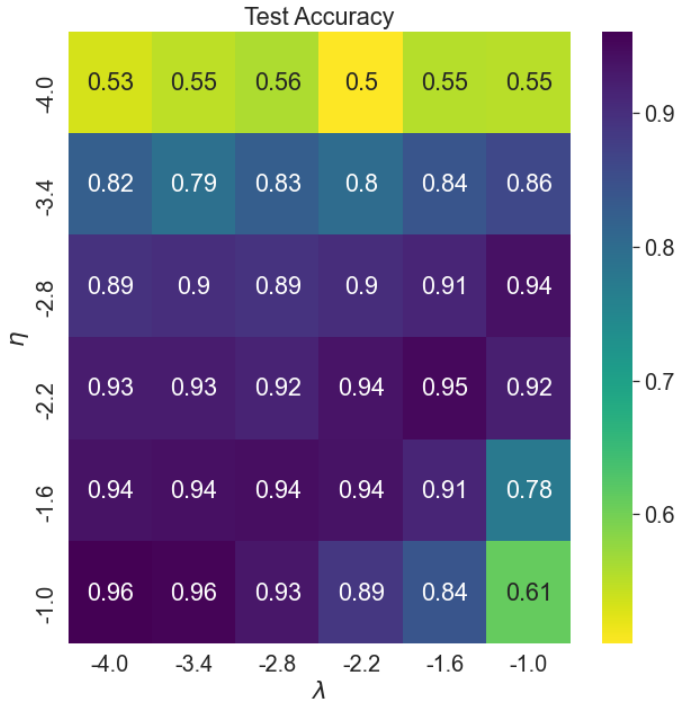

Figure 12. The test accuracy for multinomial logistic regression on the 8x8 MNIST digits. The model was trained and tested with 5-fold cross validation. We used stochastic gradient descent with 100 epochs and a batch size of 10.

We also used our feed forward neural network for classification. The best performing parameters we found were one hidden layer with 40 nodes and sigmoid activation. The output layer of course used the softmax activation function. We used stochastic gradient descent with 100 epochs and a batch size of 10.

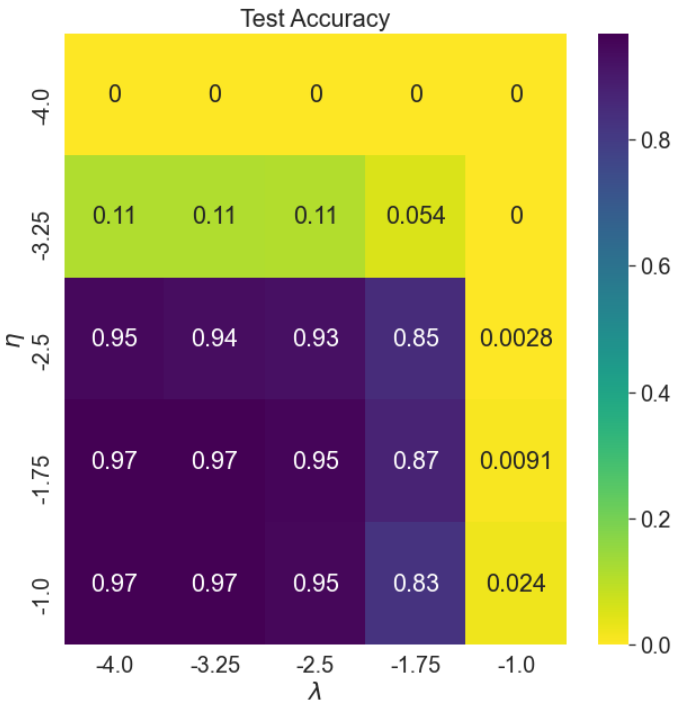

Figure 13. The test accuracy for our feed forward neural network fit to the 8x8 MNIST digits. The model here used one hidden layer with 40 nodes and sigmoid activation. The output layer used the softmax activation function, and the cross entropy cost function. The model was trained and tested with 5-fold cross validation. We used stochastic gradient descent with 100 epochs and a batch size of 10.

Finally, we used the SGDClassifier from sklearn to compare a classification model from a popular python library with our implementations. We used the SGDClassifier for logistic regression with an L2 penalty. The test accuracy scores found with 5-fold cross validation for different learning rates and regularization parameters is shown in figure 14.

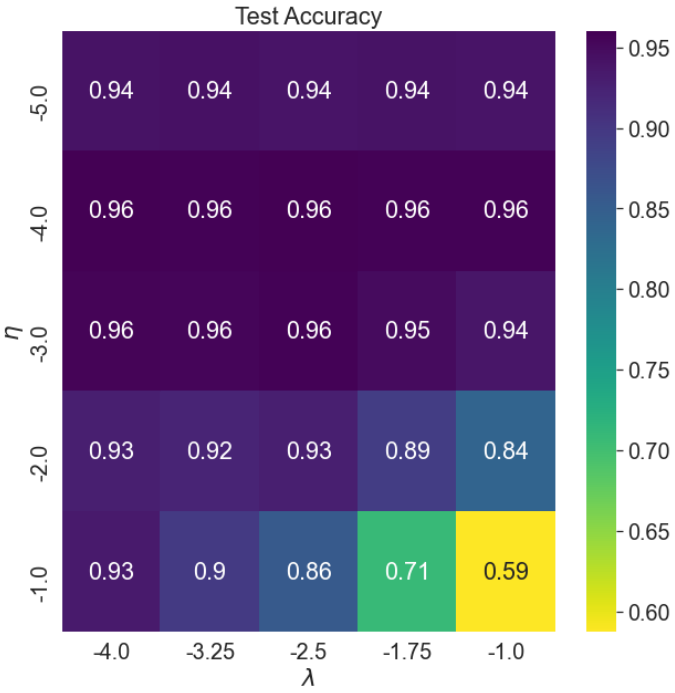

Figure 14. The test accuracy for the sklearn SGDClassifier fit to the 8x8 MNIST digits. The model was trained and tested with 5-fold cross validation. The built-in gradient descent method was used.

IV. DISCUSSION

A. Regression

The very best model for fitting the Franke function was the analytical ridge regression model. It is no surprise that it outperformed the ridge regression with SGD, as that model tries to iteratively find the minimum which the analytical ridge regression finds directly. It is also worth noting that the smallest possible λ did not give the best fit, which also should come as no surprise, as ridge often outperforms OLS, which is just ridge regression with $\lambda = 0$.

Our neural network did not outperform analytical ridge. This might be because we performed a more fine grid search of the parameter space for ridge, which might have allowed for an unlikely good fit to appear. It might also be that we did not find an especially good network structure in our limited search. More iterations or a different batch size might also be the solution. We would expect our neural network to be able to outperform ridge regression, as the ridge model simply transforms the input into clever weights (x^2y, xy^2, etc.) and finds the best coefficients to transform those weights into the correct output. A neural network should be able to generalize this behavior, with a large array of possible clever transformations and optimization of weights available. For this problem however, the function we are trying to fit is not very complicated, and a simple polynomial is a very good approximation. The neural network did not manage to capture the behavior of the Franke function as well, without overfitting the data, due to the small number of data points we used in this project. This was shown in our results for 1000 data points, where the neural networks performance was closer to the analytical ridge when only the number of data points changed.

The same things said about our neural network can be said about the tensorflow/keras model. With the right parameters, or with more data, it should be able to outperform both Ridge and our very basic implementation of a feed forward neural network and stochastic gradient descent.

B. Classification

Our feed forward neural network gave the best model for classifying the 8x8 MNIST digit images. This was expected, as multinomial logistic regression is the same as a neural network with no hidden layers and a softmax activation in the output layer, together with a cross entropy loss function. Adding more flexibility to this model can only open the possibility for improvement. The improvement we see in figure 13 (neural network) over figure 12 (logistic regression) and 14 (sklearn), is only 1%, from 96% correct to 97% correct. On the other hand, given the difficulty of the task of recognizing digits which might have been drawn very poorly and considering the bad resolution of the images, that 1% is still a notable improvement, as the room for improvement is so small.

V. CONCLUSION

We have evaluated analytical ridge regression, ridge regression with stochastic gradient descent, and a feed forward neural network on the regression problem of fitting the Franke function with a small number of data points.

We found that the analytical ridge regression found a fourth order polynomial which fit the Franke function quite well with a mean squared error (MSE) of 0.014. Ridge regression with stochastic gradient descent (SGD) was only able to find a polynomial which resulted in a MSE of 0.02, as it tried to iteratively minimize the function analytical ridge minimized directly. Both the feed forward neural network we implemented, and the one given by tensorflow/keras were unable to fit the data better than analytical ridge, due to overfitting the data with the small number of datapoints, and due to our limited search of models. Our neural network acheived a MSE of 0.015, which is still much better than ridge with SGD.

We have also evaluated our implementation of logistic regression, sklearns implementation of logistic regression

and a feed forward neural network on the classification of classifying 8x8 pixel greyscale images of handwritten digits from the MNIST data set. Both logistic regression methods achieved a test accuracy of 0.96, but the added flexibility of the neural network allowed it to improve on this with a test accuracy of 0.97.

In summary, we have found that the neural network outperforms ridge regression when there is no analytical solution for the minimum of the cost function, but that the analytical ridge model will give very good models for problems where polynomials are a good approximation to the function at hand. For classification, neural networks are preferred over logistic regression due to the increased flexibility of the model.

Future improvements to our testing and analysis could include a more thorough search of model structures for the neural network, in addition to a closer look at how the number of data points changes the performance of the models. Different gradient descent methods could also be tested, together with different batch sizes or number of epochs. Different data sets where the classical methods perform poorly would also be of interest, as the improvement offered by neural networks could be more substantial.

[1] "Github repository with code and results: https://github.com/KarlHenrik/FYS-STK-Gruppe/tree/master/Project2," (10.11.2020).

[2] R. T. Hastie and J.Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition. Springer Series in Statistics* (Springer, New York, 2009) pp. 389–414.